

Arm cross development tools

- the GNU C compiler, binutils and glibc can be configured to target the arm series of microprocessors
 - Raspberry Pi uses an arm11 processor
 - processor runs at 700Mhz
- cross development tools are much preferred to slow native tools

Configuring tools

- we have to build/install following components:
 - binutils
 - unpack_headers
 - gcc
 - newlib
 - glibc

Ordering configuration

- build the cross development binutils
 - arm-linux-elf-as, arm-linux-elf-ld, etc
- unpack the kernel headers from arm-linux
 - contains function prototypes and system call definitions
- now build a cross C compiler
 - arm-linux-elf-gcc

Continuing the tool chain build

- the cross compiler, assembler and headers complete
 - now need minimal libraries
- build crt0.o
 - found in newlib package
- now build cross glibc
 - contains open, close, read, strcat, printf, etc

Detail: headers

- are initially unpacked into: `/usr/local/arm-linux-elf/include`

- defines system calls

```
#define SYS_open __NR_open
#define SYS_read __NR_read
#define SYS_write __NR_write
#define SYS_close __NR_close
```

- taken from: `bits/syscall.h`

Use of system headers

- used so that `gcc` can build a library of functions
 - each of which maps onto a system call

```
int read (int fd, void *ptr, int len)
{
    return syscall(SYS_read, fd, ptr, len);
}
```

crt0.o

- required as it is the first piece of user code executed
 - this code calls your `main` function
 - the source to this is sometimes assembler and sometimes C
- duty is to set up the arguments for `main` and environment for `main`

crt0.c for the Raspberry Pi

```
#include <stdlib.h>
extern char **environ;
extern int main(int argc, char **argv, char **envp);
void _start(int args)
{
    /*
     * The argument block begins above the current
     * stack frame, because we have no return
     * address. The calculation assumes that
     * sizeof(int) == sizeof(void *). This is
     * okay for i386 user space, but may be
     * invalid in other cases.
     */
    int *params = &args-1;
    int argc = *params;
    char **argv = (char **) (params+1);

    environ = argv+argc+1;
    exit(main(argc, argv, environ));
}
```

Hello world

- remember hello world might be written:

```
#include <stdio.h>

int main (int argc, char *argv[],
          char *environ[])
{
    printf("hello world\n");
    return 0;
}
```

- many applications ignore the third parameter to main!

Cross glibc (C libraries)

- required as they provide: `printf`, `open`, `read`, `close`, etc
- they will in turn perform system calls and utilize the arm syscalls in `#include <syscall.h>`
- C libraries are extensive and take longer to build than the linux kernel!
- once all these pieces are installed we can build any C program (which only uses `libc`).

C compiler driver

- the C compiler driver will perform a number of activities for users
 - preprocess the C source
 - compile the preprocessed source
 - link the object files and libraries to form an executable
- examine this with:

```
$ arm-linux-elf-gcc -v hello.c
cc1 -lang-c ... hello.c -o /tmp/ccuvJUp0.s
as -o /tmp/ccsay5Hn.o /tmp/ccBNqUFj.s
collect2 ... crt0.o -lgcc -lc /tmp/ccsay5Hn.o
```

Building more compilers

- the gcc package and associated front ends can be combined to produce a number of compilers "out of the box"
 - C, f77, ADA, Java, C++
 - a few others are available from elsewhere:
 - Modula-2
 - Pascal

Building GCC as a cross compiler

```

$ tar xzf gcc-version.tar.gz
$ mkdir build-gcc
$ cd build-gcc
$ ../gcc-version/configure --enable-languages=c,c++,gm2 \
  --prefix=/usr \
  --infodir=/usr/share/info \
  --mandir=/usr/share/man \
  --target=arm-linux-gnu \
  --disable-nls \
  --disable-shared \
  --enable-long-long \
  --without-included-gettext \
  --with-dwarf2 \
  --disable-libssp \
  --build='dpkg-architecture -qDEB_BUILD_GNU_TYPE' \
  --host='dpkg-architecture -qDEB_HOST_GNU_TYPE' \
  --disable-mudflap \
  --disable-libmudflap \
  --disable-threads

```

- do *not* do this on a production system!
- note that if you perform this you will be placing binaries into your production system `/usr/bin`
- far, far better to create your own debian package and perform the build in a `chrooted` environment
- or alternatively obtain an account for a virtual machine on: `mcgreg-xen`
- check out `pbuilder`

Building glibc

- in principle these are the instructions, you need to download, unpack the source code
 - be prepared to possibly apply some patches
- point your `PATH` to the location of the cross compiler `arm-unknown-linux-gnu-gcc`
 - and the cross assembler, linker, archiver: `arm-unknown-linux-gnu-as`, `arm-unknown-linux-gnu-ld`, `arm-unknown-linux-gnu-ar`
- then configure and make the library

Building GCC as a cross compiler

Crosstool

- the interested reader could check out `crosstool`
- a script which contains information about which releases of `gcc`, `binutils`, `glibc` work well together

